OMNIMONXL (TM)
USER'S GUIDE

by

David Young, CDY Consulting

SYSTEM REQUIREMENTS: ATARI 800XL Home Computer

*** ATARI and ATARI 800XL Home Computer are trademarks of ATARI, Inc.
*** OMNIMONXL is a trademark of CDY Consulting.
*** OMNIMONXL program and manual contents Copyright 1984 CDY Consulting

AUTHOR'S EARNEST ENTREATY

I have done my best to offer here a quality program at a reasonable price. This is my livelihood. Please do not make copies of this program for any reason other than personal backup. Thank you.

INTRODUCTION

Greetings fellow ATARI Home Computer owner. I am sure you are just as proud of your system as I am of mine and enjoy buying accessories to extend its power and convenience. From that point of view, OMNIMONXL is one of the most powerful additions you can make to your computer. Any serious ATARI owner will find it indispensable after using it for the first time.

OMNIMONXL is a resident machine language monitor which, once installed, is always available to you. What that means is that you never have to load it and you can call it up no matter what program happens to be running at the time. Once running, OMNIMONXL gives you complete control over your computer. This includes the ability to easily examine and modify memory or the 6502's registers, to dump data to a printer, and to read and write to the disk drive(s) without DOS. It also has a complete set of debugging tools including a disassembler, single step, and a unique JSR function for testing out subroutines. And all of these features are available to you at any time, no matter what program is running, simply by pressing SYSTEM RESET along with either the OPTION/SELECT or SELECT button!

If you have been wanting to learn assembly language programming, OMNIMONXL can make it a very pleasant experience. Since it is ROM resident, you can always get back to OMNIMONXL even if your program hangs up in an infinite loop or if the system is locked up. You can even call OMNIMONXL at critical points in your program to examine things before continuing execution.

GETTING STARTED

After installing RAMROD XL in your computer (if you have not done so, see RAMROD XL INSTALLATION INSTRUCTIONS), you should be able to powerup your computer as usual. To enter the OMNIMONXL program, hold down the OPTION/SELECT keys (press OPTION and SELECT simultaneously) and press SYSTEM RESET. This method of entering OMNIMONXL will cause a

warmstart up to the point that the application program would normally be given control. Instead OMNIMONXL takes control and you should see the OMNIMONXL header written across the top of the screen:

```
David Young OMNIMONXL (C)1984

  PC NV-BDIZC AC X  Y  SP
AF24D   73    00 09 3F 1FF
```

The contents of the CPU registers are printed, in this case reflecting the state of the RESET sequence of the OS rather than that of the application program. Later we will discuss another method of entering OMNIMONXL which preserves the PC and CPU registers of the application program. When you are ready to exit OMNIMONXL, hold down the START button and type RETURN. This will cause the warmstart to go to completion, giving control back to the application program.

There are a few important things to point out before proceeding:

1) All numerical input and output is done in hex.

2) Parameters are delimited by a space or other non-hex character.

3) The command being processed will be aborted if an illegal parameter is encountered or if a necessary parameter is not supplied.

4) It is not necessary to retype a command if it is already present on the screen. Just position the cursor on the same line, make changes if you wish, and type RETURN. All the normal ATARI editing commands are available.

5) The processing of most commands can be stopped by holding down the START button. This allows you to terminate a long listing, search or single step. Use CTRL-1 to temporarily halt and restart a listing.

DISPLAY MEMORY: D (start addr) (end addr)

This command is used to view data in memory in either hex or character format, depending on the current data format (see TOGGLE). In hex format the data is output to the screen as 1 or more lines of 8 hex bytes separated by spaces. In character format the data is output as 1 or more lines of 24 byte character strings. On each line, the address of the first byte precedes the data.

In either data format the letter 'A' is appended to the start of each line. This in fact represents the ALTER MEMORY command (see the following command description). The effect is that, once you have used 'D' to display part of memory, you can alter any byte(s) by simply positioning the cursor, typing the change(s), and hitting RETURN. You must type RETURN on each line that you alter for the change to take effect. Also, the current data format must match the way the data was represented on the line. One other limitation in the character mode is that a line containing the character representing $9B is not alterable past that character. If you wish to alter a line after a $9B character, redisplay the line starting just past it.

To display memory, type 'D' followed by optional start and stop addresses. You can display up to 32K bytes ($8000) with one command. If you omit the stop address, only a single line of data will be printed. If you omit the start address, the next logical line of data will be printed (either the next 8 or 24 bytes of memory, depending on the data format). One last convenience is that once you have used the 'D' command, OMNIMONXL will default to that command if you just type RETURN. This allows you to scroll through memory by holding down the RETURN key. This default will remain in effect until one of the other 'persistent' commands (R or X) are used, at which time they will become the default.

TOGGLE DATA FORMAT: T

As mentioned previously, all numerical data is represented in hex. However, when dealing with ASCII text it is more convenient to work in character format. The TOGGLE command (T) is used to switch between hex and character format. It affects three commands: ALTER MEMORY (A), DISPLAY MEMORY (D) and SEARCH MEMORY (S). Other commands are unaffected by the current data format.

To switch data formats type 'T' (RETURN). Upon first entering OMNIMONXL the data type defaults to hex.

ALTER MEMORY: A addr byte byte ...

This command is used to change 1 or more contiguous bytes of memory. You can type the change either as hex bytes (separated by spaces) or as ATASCII character strings, depending on the current data format (see TOGGLE). While it is possible to use the 'A' command by itself at any time, it is not recommended. To display the area of memory first with the 'D' command and then to position the cursor and make the change is much safer (see DISPLAY MEMORY). This way you not only verify that the memory at that location is the memory you intended to change, but also that the current data format is compatible with the data you are typing.

To use the ALTER MEMORY command, type 'A' followed by an address. Use a space to separate the address and the data and then start typing the data. If in hex format, type hex bytes delimited by spaces. If in character format, type a continuous character string. Terminate the command with RETURN. At that point the indicated changes will be made. The command line can be as long as you like or until the computer squawks.

SEARCH MEMORY: S addr byte byte ...

Searching is something that computers do very well and OMNIMONXL has a very nice search function that works in either hex or character mode. It will scan memory for any sequence you specify and display it in a manner similar to the DISPLAY MEMORY command every time it is found. This means you can alter any occurrence of that sequence by simply positioning the cursor, typing the change and hitting RETURN (see DISPLAY MEMORY).

To use the SEARCH MEMORY command, type 'S' followed by the address

where you would like the search to begin. Then type a space followed by the search sequence. This will be hex bytes separated by spaces in hex mode or a character string in character mode (see TOGGLE). The search will begin when you hit RETURN. The search sequence can be any length up to the limit of the ATARI terminal input buffer. Even though it only takes a few seconds to search all of memory, a search can be aborted by holding down the START button.

Hex Conversion/Arithmetic: H = (= oper) (= oper) ...

This is a versatile command for converting hex to decimal and vice versa. It will also do hex arithmetic by allowing you to specify an additional hex number followed by an operand. The operands are + (add), - (subtract), * (multiply), and / (divide). For example, say you wanted to calculate the number of single density sectors required to write an 8K block of memory to disk:

```
 H C000 A000 - 80 /
$C000=49152
$2000=2048
$0040=64
 W1 A000 40
```

$C000 is first converted to decimal, then $A000 is subtracted and the difference is printed out, and finally the result is divided by $80 and the quotient is printed out.

The rules of the Hexadecimal Arithmetic command are:

1) The first number may be in hex or decimal with a decimal number terminated by a non-hex character (i.e., 256T=$100).

2) Every number after the first must be in hex. If you need to use a decimal number, convert it to hex first.

3) The divide operand (/) rounds down for fractions less than .5 and up for fractions greater than .5.

PRINTER ON/OFF: P

If you want a hardcopy record of your OMNIMONXL session, you can use the 'P' command to cause anything being output to the screen to be echoed to the printer. In character mode, inverse video characters are printed as normal video and unprintable characters are translated to dashes (-). Otherwise, everything on the screen will show up on the printer. There is even a special single step mode (see EXECUTE) that will trace through a program while outputting only to the printer and not to the screen. This is useful for programs that use screen modes other than GRAPHICS 0.

The 'P' command is a toggle function. Typing it once will enable output to the printer and typing it again will disable output. If the printer is not turned on or selected, the message 'I/O ERROR' will result. Care should be taken if the printer is enabled while reading or writing to the disk (see READ DISK or WRITE TO DISK).

The trace turned on by the P command can be redirected to any output device. If it is desired to output to something other than the printer, store the device specification someplace in memory and point $125 (PBUFAD) to that location. The P command will then open up an I/O channel to that device and the next P command will close it. For example, if you were to store 'D:TEMP ' (notice the blank used as a terminator) at $600 and store 00 06 at $125, the P command would open up a disk file (assuming of course that an FMS is in memory).

DISK INPUT/OUTPUT

Anyone who owns my disk utility DISKSCAN knows how useful it is to be able to edit raw sector data on a disk. One of my goals in designing OMNIMONXL was to incorporate some of the features of DISKSCAN. Imagine, a resident mini-DISKSCAN!

Well, the end result has far exceeded my expectations. With OMNIMONXL you can not only read and write individual sectors, but multiple sectors to and from anywhere in memory. And it not only works in sequential mode, but it can also follow sector links. In fact, you can read in an entire DOS file from a disk without even booting up DOS! And the frosting on the cake is that OMNIMONXL works equally well in single or double density, a dream come true for the growing number of double density drive owners.

LINK/SEQ MODE & DRIVE =: L (drive=)

When you first enter OMNIMONXL, the program assumes that you wish to talk to drive =1 and that the sector mode is sequential. If you wish to address other drives or follow sector links, use the LINK command to put OMNIMONXL in the correct mode. The LINK command is actually two commands in one. When used by itself (without a parameter) 'L' means to toggle from sequential to linked mode or vice versa. When followed by a drive = (1-4), 'L' means to switch the drive ID to the specified drive. From that point on, all disk I/O will be directed to that drive.

To toggle between the sequential and linked sector modes, type 'L (RETURN)'. To direct disk I/O to a different drive, type 'L' followed by the drive = and RETURN.

READ DISK: R (sector=) (buffer addr) (= sectors)

The READ DISK command is one of the most powerful, user friendly functions of OMNIMONXL. It can be used to read one or more sectors, either sequentially or linked, from any disk drive, single or double density. We will start out by using the READ DISK command to read one sector at a time. You will find it behaves somewhat differently when operating on more than one sector at a time.

To read a single sector into memory type 'R' followed by the sector =, buffer address and RETURN. From that point on OMNIMONXL will assume that buffer address for subsequent disk I/O. Once you have the sector in memory you can operate on it with any of the other OMNIMONXL commands including DISPLAY, ALTER, SEARCH, DISASSEMBLE, etc. One convenient feature is that, after a READ DISK command, OMNIMONXL will assume the

buffer address if you use 'D' or 'X' without a start address. (Try typing 'D (RETURN)' after reading a sector into memory).

Now, if you wish to read the sector which logically follows the last sector read into memory, type 'R (RETURN)'. In sequential mode, the next physical sector on the disk will be read. In linked mode, OMNIMONXL will reference the sector link of the current sector to determine the next sector to read. In either case, the new sector will be read into memory at the SAME buffer address, overlaying the old sector.

NOTE: IF THE PRINTER IS ENABLED WHILE READING SINGLE SECTORS, THE SECTOR = AND BUFFER ADDRESS MUST BE SPECIFIED EACH TIME. This is because the printer and disk share the SIO DCB (Device Control Block).

Ready for a couple more examples of user friendliness? One is that the 'R' command, like 'D' and 'X', is a 'persistent' command. That means that once you use the 'R' command, OMNIMONXL will default to that command if you just type RETURN. This default will remain in effect until one of the other persistent commands are used. What this means is that you can read through an entire file (or disk, if in sequential mode) by reading the first sector and then simply holding down the RETURN key. One other convenience is that OMNIMONXL will not read past the end of file if it is in linked mode. Thus, if you were reading through a file as suggested above, simply hold down the RETURN key until 'EOF' is printed. At that point, the last sector of the file is at the buffer address. You are free to add something to the end of the file (perhaps an autorun vector) and then to write the sector back out with the WRITE SECTOR command.

Reading multiple sectors is somewhat different from reading single sectors. For one thing, the sector #, buffer address, and sector count must be specified each time. The other difference is that, instead of consecutive sectors overlaying each other, the buffer address is incremented between sectors so that the disk data fills memory. The exact amount by which the buffer address is incremented depends on the sector mode and the density of the drive. The effect is that in sequential mode all bytes of the sector are preserved, while in linked mode the sector links are overlayed. This is a desirable feature if you want to read an entire DOS file into memory.

An example may be of help. First, put OMNIMONXL in character mode with 'T' and sequential mode with 'L'. Now type 'R 169 6000 8'. This will read in the 8 sectors of the directory into the buffer at $6000. Now type 'D' followed by several RETURNs. You will be able to read the names of the files on the disk. Choose the filename of a short file, put OMNIMONXL in hex mode with 'T', and use 'D addr' to display the 5 bytes just prior to the filename. The first byte is the status while the next two are the size of the file and the next two are the start sector. (You can more easily determine the start sector and file size with the 'G' command.) Now put the program in linked mode with 'L'. Read in the entire file with 'R (start sector) 6000 (file size)'. Type 'D' and hold down the RETURN key to scroll through the data of the file.

One final convenience is that each time a single sector is read, its # is printed along with the buffer address. This also occurs when

multiple sectors are read, but only when the printer is on or if you hold down the OPTION switch during the operation. Thus, if you read in an entire file with the printer on, you get a sector map of that file. Now if, while inspecting the file in memory, you find that you wish to make a change to the file on disk, you can compare the buffer address to the sector map of the file to determine the sector where that piece of data resides. Then you can use 'R sec=' to fetch that sector, make the change, and use 'W sec=' to store the sector back on disk.

For those of you with Happy drives, there is one extra feature which you may find handy. If a Happy drive is read or written to with a sector = of $800 or greater, the Happy drive treats the sector = as an internal buffer address ($800-$13FF). On multiple reads or writes the sector = is incremented by $80 each sector. Usage of the RAM buffer is as follows (compliments of David Milligan of Surrealistic Software):

$800-$A7F - RAM area for program uploading, as in programming the drive to do something.
$A80-$AFF - RAM area used by the onboard O.S. as a scratchpad area. Used for pointers, flags, etc.
$B00-$13FF -RAM area for track reads and writes.

WRITE TO DISK: W (sector =) (buffer addr) (= sectors)

The WRITE TO DISK command allows you to write one or more sectors worth of memory out to disk. The one big difference between it and the READ DISK command is that it only works in the sequential mode. That means that it will not create a DOS file, i.e., it will create neither a directory entry nor sector links. If you do wish to create a DOS file out of memory it is best to use the BINARY SAVE option of DOS. However, it is not always possible to get DOS into memory without losing your data. In this case, OMNIMONXL may be the only way save your data. If you do wish to create a DOS file out of the memory you have written to disk with OMNIMONXL, use the technique described under the 'G' command.

IMPORTANT: Use a scratch disk when writing multiple sectors worth of memory to disk with OMNIMONXL. The program pays no attention to data already on the disk and may overlay it.

The primary purpose of the WRITE TO DISK command is to support the modification of one sector at a time. A typical scenario is as follows:

> Turn the printer on, read a file into memory as described in READ DISK, and turn the printer off. Search the file to find the data to be changed. Compare the address of the data in memory to the sector map created while the file was being read in. Read that particular sector into memory with 'R sec='. This insures that you not only have the data of the sector but also the sector link. Then alter the sector and write it back out with 'W sec='.

Be careful when omitting the sector = because the default has already been incremented in anticipation of the next READ DISK. In fact, the only safe time to omit the sector = is when that sector is the last one of a linked file.

Binary Load / Directory: G (file spec) (addr)

The versatility and convenience of this command is really quite amazing. First of all, it will load any binary load file from a single or double density ATARI DOS compatible disk (2.0S, 2.0P, MYDOS, OSA+2.0, etc.). This includes files that even DOS cannot load because they load on top of it. Secondly, as it searches the disk directory for the specified file, it prints out the filenames, file sizes and start sectors in hex. For example, put a disk with a binary load file into drive #1. Now type:

G D:filename

Pretty nice. Here are the rules for using 'G':

1) The file specification is similar in format to that of DOS. For example, 'D:FILE', '1:FILE', and 'FILE' all are equivalent to 'D1:FILE'.

2) If you do not give a file specification, 'G' will search the directory and not find a match but in the process will print out the entire directory. Thus 'G(RETURN)' will give the directory of drive #1 while 'G2:(RETURN)' yields the directory of drive #2.

3) Another nice feature of 'G' is that it will print out the load vectors of a binary load file if you hold down the OPTION key while the file is being loaded. It will print out each load vector and pause before it loads the data to satisfy that load vector. Pressing SELECT will load the data and print the next load vector. If you wish to terminate the load, press START. Try loading several birary load files while holding down the OPTION key and you will be surprised at how complicated some of them are (the ATARI Macro Assembler for instance).

4) One final option concerns the sector buffer address which 'G' uses while it is loading the file. If you do not specify an address, $400 is the default. If the binary file happens to load into this area ($400-47F for single density or $400-$4FF for double), you may specify another buffer address in hex which does not interfere with the load.

By the way, once the binary load has started, 'G' uses zero page locations $43-49. These are locations reserved for use by DOS, in particular during a binary load. Thus, 'G' should load anything that the 'L' option of DOS does. However, some files may not load correctly if you use the console switches as described in rule #3 above because the resources used to print out the load vectors may interfere with the load itself.

Once you have a binary load file in memory you can create a boot disk by switching over to sequential mode and writing the program back out to a disk starting at sector 1. You will need to leave 6 overhead bytes at the beginning of the first sector. See the ATARI OPERATING SYSTEM USER'S MANUAL for details on the boot process.

The converse of this process would be to create a binary load file from a boot record. This can be done by first booting up DOS, going to OMNIMONXL, and reading in the boot record in sequential mode to a convenient place in memory. Then you would exit back to DOS and do a BINARY SAVE on that portion of memory. Then you may have to use OMNIMONXL to change the load vector at the beginning of the file to make it load in at the correct place. In fact, you may have to use OMNIMONXL to load the file if it loads on top of DOS. The same technique can be used to make a binary load file out of a cartridge but you will have to append a few load vectors to get the program going. For example, the following 3 load vectors should be appended to the end of BASIC: 6A 00 6A 00 90 E2 02 E3 02 F6 F3 E0 02 E1 02 00 A0.

How do you go about appending these load vectors? One easy way to add a few bytes to the end of a file is as follows:

1) Find the last sector of a file by determining the start sector, putting OMNIMONXL into the linked mode, reading it into memory with the 'R' command and holding down RETURN until 'EOF' is printed out.
2) Determine the last data byte of the sector by looking at the byte count (the last byte of the sector).
3) Just past the last data byte add the new bytes. Then increase the byte count to reflect the appended bytes and write the sector back out with 'W(return)'.

If you find this discussion confusing, read the tutorial on binary load files at the end of this document.

SEVERAL WAYS TO ENTER OMNIMONXL

We have seen how to enter OMNIMONXL by holding down OPTION/SELECT and pressing SYSTEM RESET. This causes a normal warmstart followed by a jump subroutine (JSR) to OMNIMONXL. When you exit OMNIMONXL after entering it in this way (by holding down START and pressing RETURN), the warmstart goes to completion in a normal fashion. This is fine for some applications, but there is another way to enter OMNIMONXL which disturbs the program running as little as possible.

When you hold down SELECT and press SYSTEM RESET, the program running at the time is interrupted. However, instead of doing the entire warmstart, parts of it are skipped over so as to preserve the state of the system as much as possible. Specifically, the OS variables and the stack are left undisturbed. Usually this allows you to reenter the program by simply exiting OMNIMONXL in the normal fashion. For instance, you can pop into OMNIMONXL from either DOS or BASIC, execute some OMNIMONXL commands, and pop back into the interrupted program almost as if you had never left it. I say 'almost' because the OS is likely to return a bogus value if it was waiting for a keystroke when it was interrupted. For that reason it is best to hit BREAK upon return to the program. Of course, if the program makes use of any graphics other than MODE 0, it is unlikely that you will be able to successfully reenter the program without restarting it. This is also true of programs which alter the interrupt RAM vectors ($200-$224) because OMNIMONXL restores them to their original values.

There are a couple of small problems with using SELECT/RESET (instead of OPTION/SELECT/RESET) to interrupt DOS or BASIC. OMNIMONXL makes use of the SIO interrupt routines in the OS ROM by altering the interrupt vectors at $20A-$20D. This is so the printer and disk interface of OMNIMONXL will work even if DOS is not in memory. Now if you return back to DOS with START/RETURN these interrupt vectors will remain in effect. But DOS hangs up occasionally unless it is using its own special SIO handlers. If you wish DOS to restore its special vectors, exit OMNIMONXL with SYSTEM RESET. Another problem with SELECT/RESET is that MEMLO ($2E7) gets restored to $700 so that the FMS or any other program in low memory is unprotected. For that reason, it is best to exit OMNIMONXL back to BASIC with RESET. Alternatively, always use OPTION/SELECT/RESET to enter OMNIMONXL from BASIC.

Another way to enter OMNIMONXL is particularly useful for debugging assembly language programs. This is accomplished by putting 'JSR $C001' at critical points within the program. At each of these points OMNIMONXL will be entered and you will have all of its facilities available for examining the intermediate results of your program. When you are ready to continue executing your program, just exit OMNIMONXL with START/RETURN. There are some restrictions on this technique however, specifically special graphics and time critical I/O.

Yet another way to enter OMNIMONXL is from BASIC with a 'X=USR(49152)'. You can exit back to BASIC in the usual manner (START/RETURN).

It should also be pointed out that OMNIMONXL will be entered automatically if a 6502 BRK instruction (0) is ever executed. Thus, you can set a breakpoint anywhere in your program by storing a 0. When you pop into OMNIMONXL after executing a BRK instruction, you should restore the original instruction and subtract 2 from the PC. Now you can continue executing your code when you exit OMNIMONXL.

CPU REGISTERS: C

You will notice that, upon entering OMNIMONXL, the 6502's internal registers are printed out with the following heading:

```
PC NV-BDIZC AC  X  Y  SP
```

The meanings of these headings are pretty self-explanatory except for 'NV-BDIZC'. These are the individual bits of the status register spelled out. Thus, this is a snapshot of the state of the CPU just prior to entering OMNIMONXL. The PC (program counter) is pointing to the next instruction to be executed. The program will continue executing at this point when you leave OMNIMONXL with START/RETURN.

The CPU state can be examined at any time with the CPU REGISTERS command 'C'. In addition, the CPU state can be changed by simply positioning the cursor over the value, typing the change, and hitting RETURN. The new values for the registers will be in effect when you leave OMNIMONXL to resume execution of the suspended program. The only CPU register that cannot be changed directly is the stack pointer. This can be changed only by the PUSH STACK (+) and POP STACK (-) commands.

One application for the 'C' command is to GOTO anyplace in memory. This is accomplished by altering the PC to point to the address where you wish execution to resume when you press START/RETURN. Typically this might be back to DOS, whose address can usually be found by looking in location $000A (DOSVEC).

Another area of interest is the stack. Remember, the stack pointer always points to the next FREE entry. All the values between the stack pointer and $1FF are typically return addresses of nested subroutine calls. This, in fact, is a vertical cross section of the execution history of the program. This is extremely useful for finding your way around in a program you wish to modify in some way. If you wish to to locate the part of a program which is performing a certain function, just start the program executing that function and press SELECT/RESET. Because the stack is preserved with this method of entering OMNIMONXL, you can tell where the program is and where it has been by noting the PC and the return addresses on the stack. Another way of locating a certain piece of code is to search ('S') for a particular address it might reference.

PUSH STACK: + byte byte ...

The PUSH STACK command is for adding bytes to the stack and thereby increasing the stack pointer (which grows downward in the 6502). These bytes will be available to the code pointed to by the PC when OMNIMONXL is exited. Notice that the first byte after '+' is the first one to be pushed onto the stack.

Please note that the stack pointer displayed with the 'C' command is not the ACTUAL stack pointer while OMNIMONXL is running. OMNIMONXL uses the stack for its own purposes and is actually nested somewhat deeper. It is not wise to make changes directly to the stack unless you use PUSH STACK or POP STACK.

POP STACK: -

The POP STACK command takes bytes off of the stack one at a time and decreases the stack pointer (which actually increases in value).

DISASSEMBLE MEMORY: X (start addr) (stop addr)

Just as it is possible to display memory in hex or character format, it is also possible to translate 6502 machine code to assembly language. OMNIMONXL does this in a handy fashion by printing out the object code along with the instruction. Once again, it is possible to change the object code (to the left of '*') by positioning the cursor, typing the change, and hitting RETURN. Another convenience is that the value at the address specified with indirect addressing modes (without regard to the index register) is printed in parentheses.

Just like the 'R' and 'D' commands, 'X' is 'persistent'. Once you have disassembled one or more instructions, you can continue disassembling simply by holding down RETURN. This will remain in effect until 'R' or 'D' are used. The disassembler can be aborted at any time by pressing the START button.

Assembler: Y addr instruction

This will not take the place of your 2 pass assembler. 'Y' is a line assembler, meaning that you type in a line of assembly language and it will convert it to machine code immediately. If you have ever had to hand patch 6502 object code you will know how handy this function can be. It saves you from having to look up the opcodes in a table. Also, sometimes it is desirable to write a quicky routine to try something out. For example, say you wanted to see what the key codes are that get stored in location $2FC whenever you press a key. To do so you could type in the following commands:

```
Y 600 LDA $2FC   (get key code)
  CMP #$FF
  BEQ $600   (no key pressed?)
  STA $610   (store new key code)
  RTS
(just hit RETURN to terminate assembler)
```

Notice that you need to type 'Y addr' only once. From that point on the assembler will prompt you for the next instruction. To execute this routine:

```
J 600
(START/RETURN)
(press a key)
D 610   (to see what the key code is)
```

Here are the rules of the assembler:

1) To enter the assembler, type 'Y addr instruction' where 'addr' is the starting address in hex and 'instruction' is a legal 6502 assembly language instruction.

2) To exit the assembler just type RETURN when prompted for the next instruction.

3) If you want to make a change to an assembly language instruction which is already on the screen, just move the cursor up to that instruction (it must be the line with the 'Y' at the beginning), make the change, and hit RETURN. This is valid whether or not you are currently in the assembler.

4) When specifying operands, hex numbers are preceded by '$' and decimal numbers are not.

5) Branch instructions (BEQ, BNE, etc.) are handled quite easily. The operand can be specified as an absolute address (in hex or decimal) and the assembler will calculate the displacement. Or, if you prefer, you can specify a displacement preceded by + or -. Thus, in the example above, the 'BEQ $600' could be replaced by 'BEQ -5' with the same result.

EXECUTE MEMORY: E (option/= steps)

The EXECUTE MEMORY command is actually a single step command in disguise ('S' is used for SEARCH). This command causes the instruction pointed to by the PC to be executed. Then the registers are printed out along with the NEXT instruction to be executed. If the step count was 1 (or not specified) then execution will stop. Otherwise it will continue single stepping through the code for the specified = steps. The maximum number of steps at one time is 31 for reasons soon to become clear.

While the low order 5 bits of the optional parameter are a step count, the high order 3 bits have special meaning. The MSB means 'step forever'. Thus, 'E 80' means 'step forever and print the trace to the screen'. Notice that the trace will also be echoed to the printer if it is enabled. Stepping can be aborted by pressing START.

Bit 6 of the parameter means 'don't print the trace to the screen'. However, the trace will still be output to the printer if it is enabled. Thus, 'E C0' would step forever without printing the trace to the screen. In combination with the printer this is useful for stepping through programs which use special graphics modes.

Bit 5 of the parameter means 'sample the results of every 32 instructions'. Thus, 'E E0' would step forever without printing to the screen and the trace would be output to the printer every 32nd instruction (if it is enabled). This is kind of a weird mode, but somebody may find a use for it someday.

One other nice feature of the 'E' command is that it will treat a call to the OS as a single instruction instead of stepping through all the code in the OS. OMNIMONXL does this by temporarily giving up control of the CPU but intercepting it on the return from the OS. However, you should avoid stepping through CIO calls to the screen editor (E:) unless printing to the screen is disabled with bit 6. OMNIMONXL considers any address above $C000 to be OS.

We have seen that the EXECUTE MEMORY command is very powerful and versatile. One restriction, however, is that it will not step through a 'SEI' instruction. If you are stepping through a program and encounter a SEI, disassemble on past it to find the 'CLI'. Just past the CLI put a temporary BRK instruction (0). Now step through the SEI. OMNIMONXL will temporarily lose control of the program but will regain it when the BRK instruction is executed. Now restore the original value to the location where the BRK was set. After subtracting 2 from the PC you are ready to continue stepping.

JSR: J addr

The JSR command is a very powerful feature for executing a subroutine and returning control back to OMNIMONXL. It can be used for testing out subroutines during the development of an assembly language program. With some care it can also be used to call the OS to, say, format a disk.

When you execute the 'J' command you will notice that the registers

are printed out but that the subroutine is not yet executed. In fact, the 'J' command does nothing more than change the PC to the specified address and push the address of OMNIMONXL on the stack to act as the return address for the subroutine. Now you are free to set up for the subroutine call by altering the registers or memory if necessary. When you are ready to actually execute the subroutine press START/RETURN. Upon return you will notice that the PC is restored to its original value but that the other registers reflect the results of the subroutine.

As an example, put a fresh disk (or one you don't mind formatting) in drive 1. With the printer disabled, store a 1 in $301, a $21 in $302, and a $80 in $303. Now execute a 'J E453' and press START/RETURN. The disk in drive 1 will be formatted and then control will be returned to OMNIMONXL.

Sometimes after interrupting a program with OMNIMONXL, you will not be able to restart it without reinitializing it. The start and initialization addresses for a program are typically at $000A and $000C respectively. Since a proper initialization routine is always a subroutine, you can use 'J (init addr)' to initialize the program. When control returns to OMNIMONXL, you need only change the PC to the start address and exit OMNIMONXL with START/RETURN to restart the program.

Move Memory: M addr0 addr1 addr2

This command is for moving blocks of code from anywhere in memory to anywhere in RAM. It matters not if the source and destination blocks overlap. The move command is very simple to use. Just supply the following addresses:

addr0 = source start address
addr1 = source end address
addr2 = destination start address

For example, say you have the following code located at $600 (you can use the 'Y' command to enter it):

```
$600 LDA $2FC
$603 CMP #$FF
$605 BEQ $600
$607 STA $60D
$60A JMP $60E
$60D NOP
$60E RTS
```

I know this is a dumb program but I wish to make a point. To move the code to $620, type: M 600 60E 620. Now use 'X' to disassemble the code at $620 and you will recognize it as the same. Now, unless the code were relocatable, you would not be able to run the code at the new location without adjusting some of the absolute addresses. Which addresses? The ones which reference locations within the address space of the program. The 'STA $60D' and 'JMP $60E' would both have to be adjusted. That is the purpose of the Relocate command ('N'), to be described next.

Verify Memory: V addr0 addr1 addr2

Compare 2 blocks of memory and print differences. The first block starts at addr0 and ends at addr1. The second block starts at addr2. Notice that the parameters were designed to complement the move command. For example, if you just moved some code with 'M', you can verify the move by simply changing the 'M' to a 'V' and hitting RETURN.

Relocate 6502 Code: N addr0 addr1 addr2 (addr3) (addr4)

Relocate will adjust code assembled to run in one location so that it will execute in another. The format is as follows:

addr0 = start of addr reference range to be adjusted
addr1 = end of addr reference range to be adjusted
addr2 = new base addr.
addr3 = start addr of code to be adjusted (default: addr2)
addr4 = end addr of code to be adjusted
(default: addr3+[addr1-addr0])

This command was designed to be very versatile but, because it has so many options, it can be quite confusing. However, since it was also designed to complement the 'M' command, its use usually requires no thinking at all. Specifically, in the example above we used 'M 600 60E 620' to move the code from $600-60E to $620. Because it has absolute memory references to the range of $600-60E, the copy at $620 will not execute properly. If we now move the cursor back up to the 'M' command and change the 'M' to an 'N' ('N 600 60E 620') and hit RETURN, the code at $620 will be adjusted to run at $620. Try it and then disassemble the code at $620. Notice the absolute memory references to the range of $600-60E have been adjusted to reflect the new base address of $620.

Now, say we wanted to adjust the code at $600-60E so that it will run at $700 but we don't want to move there to do it (perhaps you did not want to wipe out DOS, which starts at $700). In this case we must specify addr3 and addr4 as follows:

N 600 60E 700 600 60E

Notice that addr0,1 & 2 are the same as if you had moved the code to $700 first ('M 600 60E 700') but that addr3 & 4 tell 'N' that the code physically resides at $600-60E. After execution of the above command the code at $600-60E disassembles to this:

```
$600 LDA $2FC
$603 CMP #$FF
$605 BEQ $600
$607 STA $70D
$60A JMP $70E
$60D NOP
$60E RTS
```

Notice that the reference to $2FC is not modified because it is outside the address range specified by addr0,1. Also, don't worry about 'BEQ $600' because it is relocatable.

Before you try relocating a big program like a cartridge, let me point out that it will probably still take considerable work. The 'N' command cannot differentiate between 6502 instructions and imbedded program data. The only thing it can do is stop if it hits an illegal opcode and this may or may not be soon enough to avoid modifying some data it should not have (data that happened to look like 6502 code with absolute address references). It also cannot do anything about indirect references to the address range of addr0,1 (for example, indirect references to a data table imbedded within the program). Likewise, jump tables will not be adjusted. The data tables should be easy to locate and move with the 'M' command. The indirect references will probably be more difficult to find and adjust. The jump tables will have to be adjusted by hand. Thus, the 'N' command is most easily used to move relatively small routines around.

Although big programs may present a problem, I will give one other example to show how versatile the Relocate command really is. Say you wanted to move a routine from one part of a program to another. Not only would you have to relocate the routine but you would also have to use the 'N' command on the rest of the program to adjust any references to that routine. Say you had the following program:

```
$600 LDA $604,X (start of routine)
$603 RTS
$604-607        (imbedded data table)
$608 LDX #3     (this is start of program)
$60A JSR $600
$60D STA $D001,X
$610 DEX
$611 BNE $60A
$613 RTS
```

You would like to move the routine and associated data table from $600 to $614. The following commands will do this:

```
M 600 607 614 (move little routine and table)
N 600 607 614 608 617 (relocate entire program)
```

After these commands the program should look like this:

```
$608 LDX #3     (this is start of program)
$60A JSR $614
$60D STA $D001,X
$610 DEX
$611 BNE $60A
$613 RTS
$614 LDA $618,X (start of routine)
$617 RTS
$618-61B        (imbedded data table)
```

Study this example carefully. If you understand it completely then you have mastered the 'N' command. You will find it (along with 'Y', 'M' and 'X') a big help in patching programs for which you don't have source code or don't want to take the time to reassemble.

Fill Program Buffer: F addr

If you find yourself using a certain sequence of OMNIMONXL commands frequently, you may want to program the monitor to execute them automatically. Likewise, if you are debugging or analyzing a program, you may want the monitor to remember the sequence of commands so that you can duplicate them at a later time. The 'F addr' command tells OMNIMONXL to start saving all your commands at the specified address (the program buffer). It will continue to save your commands until you execute an 'F0' to tell it to quit. Then the 'O addr' command is used to execute from the program buffer at any later time.

The format of the 'F' command is as follows:

F addr - 'addr' can be any non-ZPAGE address in RAM. It will enter the program mode at this point, remembering all subsequent commands. An addr of 0 terminates the program mode.

The program buffer should be a place in memory that will not interfere with anything else you are doing. As long as you are in the program mode the ASCII data stored in the buffer will continue to grow, eventually wiping out everything in its path if you forget to terminate the program mode with 'F0'.

Since the data stored in the program buffer is ASCII text, you may edit it if you wish. Just remember that the program data must be terminated with a hex 0. Also, if you wish to enter the program mode and append to the end of the current data rather start over, you may do so as follows:

```
 T              (to get into character mode)
 S addr F0      (addr=program buffer; find 'F0')
A xxxx F0-------(which previously exited program mode)
 F xxxx         (start filling buffer at that location)
```

One slight annoyance you will encounter in the program mode is that the result of the 'T' command depends on the the current mode (char or hex). If you use the 'T' command as part of the programmed sequence you will want to force one mode or the other at the beginning of the sequence so that you will always get the same results no matter what mode you happen to be in when you execute the sequence later. It is possible to force the hex mode with the following sequence:'A98 0'.

If the data of your command buffer is of any importance you may want to back it up occasionally to a scratch disk. This could be done as follows:

1) Find the 'F0' at the end of the buffer as in the example above and add 3 to determine the end address.

2) Subtract the start address from the end address and divide by the sector size to determine the number of sectors to write.

3) Write that number of sectors off to a scratch disk with the 'W' command. Be sure to include the buffer terminator, hex 0.

Restoring a previously saved buffer is as easy as reading the sectors back into memory with the 'R' command. If the command sequence is of lasting importance, you may want to create a binary load file by going to DOS and doing a BINARY SAVE on the command buffer. The 'G' command may then be used to fetch it at any time.

Operate from program buffer: O addr

This command is used to execute OMNIMONXL commands stored as ASCII text somewhere in memory (and terminated with a hex 0). The format is as follows:

O addr - 'addr' is the address of the OMNIMONXL commands stored as ASCII text.

Upon execution of the 'O' command, the display editor (E:) device vector at location $321 is replaced with a pointer into a special handler in OMNIMONXL which takes its input from the program buffer (which is pointed to by $98, COMPTR) instead of the keyboard. It will continue to do so until the command interpreter hits a hex 0 in the program buffer, at which point input it will revert back to the keyboard.

One thing you will notice when using the 'O' command is that only the results of the commands are printed, not the commands themselves. If you need to see the commands also, turn on the printer with 'P' prior to executing 'O'. The commands will show up on the hardcopy.

Boot Disk

Boot off of the drive selected with 'L'. You can boot off of a drive other than drive #1 only if it is a single stage boot.

Binary Load Files

This is one area that most people are a little fuzzy on. It's not surprising really, since there does not appear to be any definitive documentation on it anywhere! An exhaustive presentation will be made here even though it will be quite short.

Definition: load vector - from 4 to 6 bytes consisting of 2 optional bytes of FF FF, a 2 byte start address, and a 2 byte end address (in that order).

Example: FF FF 00 06 02 06 - The first 2 bytes are optional and ignored during the load process. The second 2 bytes are a start address of $600 and the last 2 bytes are an end address of $602.

The only time that the first 2 bytes of FF FF are required is at the beginning of a binary load file. If those bytes are not there, DOS will refuse to perform the binary load. (The 'G' command of OMNIMONXL will, however, load it gladly.) The rest of the time the first 2 bytes of FF FF, if they exist, are ignored. The only time that these 2 optional bytes should occur anywhere else but at the beginning of a file

is if 2 binary load files were appended together.

What does a load vector do? It tells DOS (or the 'G' command of OMNIMONXL) where in memory to put the 1 or more bytes which follow in the file. How many bytes is determined by subtracting the start address from the end address and adding 1. In the example above, 3 bytes would be read from the file and put in locations $600 to $602.

What happens when enough bytes have been read in to satisfy a load vector? These things will happen in this order:

1) Locations $2E2 and $2E3 will be examined. If they are both zero, goto step 2. If they are nonzero, a JSR will be made to the address contained in these locations. Upon return, zero $2E2 and $2E3 and fall into step 2.

2) If the end of file is not reached (i.e., there are more bytes in the file), another load vector is assumed to immediately follow and will be processed as previously described.

3) If the end of file (EOF) is reached, examine locations $2E0 and $2E1. If they are zero, terminate the binary load. If they are nonzero, do a JSR to the address in these locations. Upon return, terminate load.

Still confused? Let me try to simplify. If you see a load vector like 'E2 02 E3 02', you know that the subroutine at the address specified in the following 2 bytes will get executed immediately, prior to continuing the load process. If you see a load vector like 'E0 02 E1 02', you know that the subroutine at the address in the following to 2 bytes will be executed after the end of file is reached.

Now that you understand everything there is to know about binary load files, let's take a typical example: converting the BASIC cartridge to binary load file. I use this example because it is instructive and, because of the lack of copyright notice, appears to be legal. Using this technique on other cartridges could be illegal and may not work anyway due to the booby traps designed to prevent them from running out of RAM.

Let's see what it takes to get the BASIC program to run out of RAM:
-Turn on the computer with BASIC installed and pop into OMNIMONXL.
-Insert a formatted scratch disk into the drive and execute the following command: 'W1 A000 40 (RETURN)'.
-Remove the BASIC cartridge, boot up DOS and pop into OMNIMONXL.
-Because OMNIMONXL restores MEMLO to $700, we should reinitialize DOS by doing a JSR to the address at DOSINI ($C,D). For 2.0S that would be 'J1540 (RETURN) (START/RETURN)'.
-Move the screen down by storing a $90 in location $6A and doing a JSR $F3F6: 'A 6A 90 (RETURN)','J F3F6 (RETURN) (START/RETURN)'
-Read BASIC back into memory with 'R1 A000 40 (RETURN)'.
-Find the initialization address by looking at location $BFFE. Since that is $BFF9, execute 'J BFF9 (RETURN) (START/RETURN)'.
-Find the start address by looking at location $BFFA. Since that is SA000, execute 'J A000 (RETURN) (START/RETURN)'.
BASIC will now come up running. Now we want to create a binary load file to simulate the last 4 steps:

-Type 'DOS' to get to the DOS menu.
-Use the 'K' command to save BASIC as a binary load file giving it the start address of A000 and the end address of BFFF.
-Pop into OMNIMONXL and put it in linked mode. Read the first sector of the new file into $6000 (see top of page 7 if you don't know how to find the first sector of a file).
-Hold down RETURN until 'EOF' is printed out. You now have the last sector of the file in memory.
-Determine the last byte of that sector in use by looking at the byte count ($607F). Start adding the following bytes at location $6000 + byte count (we are appending to the file):6A 00 6A 00 90 E2 02 E3 02 F6 F3 00 98 08 98 20 06 98 6C FA BF 6C FE BF E0 02 E1 02 00 98.
-Increase the byte count ($607F) by $1E and write that sector back out to the disk with 'W (RETURN)'.

Command Summary

Page =

A: 3 Alter Memory: A addr byte byte ... - Used to change 1 or more contiguous bytes of memory.
B:18 Boot Disk: B - Will boot off of the selected drive.
C:10 CPU Registers: C - Used to display and alter the registers.
D: 2 Display Memory: D (start addr) (end addr) - Used to view memory data. To alter memory, position cursor and type change.
E:13 Execute Memory: E (option/= steps) - Will execute one or more instructions at a time and display intermediate results.
F:17 Fill Prgm Buffer: F addr - Teach monitor a sequence of commands for later execution with the 'O' command.
G: 8 Get File: G (file spec) (addr) - A full binary load function, single or double density. Doubles as disk directory command.
H: 4 Hex Arithmetic: H = (= oper) (= oper) ... - Hex conversion allowing addition, subtraction, multiplication and division.
J:13 Jump Subroutine: J (addr) - Go execute subroutine.
L: 5 Link Drive: L (drv=) - Select drive = and linked or sequential sector modes. All disk I/O will go to the selected drive.
M:14 Move Memory: M addr0 addr1 addr2 - Move a block of memory from anywhere in memory to anywhere else.
N:15 Relocate Memory: N addr0 addr1 addr2 (addr3) (addr4) - Adjust 6502 code that it will execute in another location.
O:18 Operate Prgm Buffer: O addr - Execute the commands stored earlier with the 'F' command.
P: 4 Printer Control: P - Screen I/O can be echoed to a printer.
R: 5 Read Disk: R (sect=) (buff addr) (= sects) - Read one or more sectors from selected disk drive into a specified buffer area.
S: 3 Search Memory: S addr byte byte ... - Search memory for and print out occurrences of a sequence of bytes.
T: 3 Toggle Format: T - Toggle between hex and character formats.
V:15 Verify Memory: V addr0 addr1 addr2 - Compare 2 blocks of memory and print out the differences.
W: 7 Write Disk: W (sect=) (buff addr) (= sects) - Write one or more sectors to disk from a specified buffer address.
X:11 Disassembler: X (addr0) (addr1) - Translate machine code into assembly language. Can be used to create a source file.
Y:12 Assembler: Y addr instr - Translate assembly language into machine code one line at a time. Useful for patching programs.